嵌入式 Forth 虚拟机架构的多任务调度算法设计与实现 *

代红兵，周永录†，安红萍，黄忠建

(云南大学 信息学院, 云南省高校数字媒体技术重点实验室, 昆明 650223)

摘 要：针对嵌入式应用领域对操作系统在重构、扩展、移植、交互、安全、高效等方面日趋苛刻的现实需求及 Forth 系统所固有的特性，采用 Forth 虚拟机技术，对基于 Forth 虚拟机架构的嵌入式操作系统关键技术进行探索，提出一种具有良好扩展和移植特性、高效精简的基于 Forth 虚拟机架构的嵌入式多任务操作系统调度算法。该算法采用了以 Forth 虚拟机指令同步的协同式多任务调度机制，缩短了任务切换时间，将上下文切换操作简化为只需保存数据堆栈指针。实验结果表明，基于 Forth 虚拟机架构的多任务调度算法发挥了 Forth 系统所固有的特性，针对特定应用，提高了效率，适合资源有限的嵌入式环境。

关键词：Forth 虚拟机；多任务；调度算法

中图分类号：TP316 doi: 10.3969/j.issn.1001-3695.2017.08.0864

Design and implementation of multitask scheduling algorithm for embedded Forth virtual machine architecture

Dai Hongbing, Zhou Yonglu†, An Hongping, Huang Zhongjian

(*Digital Media Technology Key Laboratory of Universities in Yunnan*, *School of Information Science & Engineering Yunnan University*, *Kunming* 650223, *China*)

Abstract: In embedded application fields, the operating system presentes the more stern reality demand such as reconstruction, expansion, transplantation, interaction, security, and efficient. In combination with the above features of embedded system and inherent characteristics of Forth systems, using Forth virtual machine technology, this paper explored the key technologies of embedded operating system based on Forth virtual machine architecture, and proposed an embedded multitask operating system scheduling algorithm based on Forth virtual machine architecture, which had better extension and transplantation characteristics. The algorithm adopted a cooperative multi-task scheduling mechanism with Forth virtual machine instruction synchronization, shortened the task switching time, and simplified context switching operations to simply saving the data stack pointer. The experiment results demonstrate that the multitask scheduling algorithm based on Forth virtual machine architecture develops the inherent characteristics of Forth system, presents higher efficiency for specific applications, especially benefits for embedded environments with limited resources.

Key Words: Forth virtual machine; multitask; scheduling algorithm

0 引言

Forth 语言本身就是一种过程控制语言和一种快速开发环境，而不仅仅是一个单纯的编程工具，具有很强的交互性、构造性、移植性和自扩展能力，以及高效的生成代码，甚至可以快速构造出一个实时多任务操作系统。真正将 Forth 推向操作系统领域的是美国 Kitt Peak 天文台开发的 Stand Alone Forth11[1]，随后出现的 Stand Alone Forth88[2]、Colorforth[3]、SwiftOS[4]等系统也基本秉承了基于 Forth 虚拟机（Forth virtual machine architecture，FVM）的设计思想，逐渐形成了独树一帜的基于 FVM 的嵌入式多任务操作系统（The embedded multi-task operating system based on the Forth virtual machine architecture，FVMOS）。

一直以来，FVMOS 究竟采用何种多任务调度策略始终是 Forth 界长期争论的焦点，以至于在先后形成的 FIG-Forth 、Forth-79、Forth-83、ANSI X3.215-1994、ISO/IEC 15145 :1997、FORTH-2012 等标准中均未出现相关的内容。与直接基于 CPU 处理器架构的多任务操作系统不同，FVMOS 的实现有其独特

基金项目：国家自然科学基金资助项目（61640205）

作者简介：代红兵（1963-），男，江西赣州人，正高级工程师，硕士，主要研究方向为嵌入式系统、数字电视技术；周永录（1965-），男（通信作者），高级工程师，主要研究方向为嵌入式系统（zhylu@126.com）；安红萍（1981-），女，助理研究员，主要研究方向为嵌入式系统；黄忠建（1995-），男，硕士研究生，主要研究方向为嵌入式系统.

性，若引入抢占式调度，势必会破坏虚拟机的硬件抽象，从而丧失固有的重构、扩展、移植等特性。Stand Alone Forth88 采用的就是抢占式调度，同时具备并发（仅仅是单一的终端任务）、定时和中断三种任务类型，但这种实时性保证是以数十项复杂的 CPU 现场保护与恢复以及丢失重构、扩展、移植等特性为代价的。此外，如果存在终端任务，这种中断驱动的强制调度还会严重干扰和打乱 Forth 系统特有的在线交互过程。那么，究竟有没有一种既能发挥 FVM 特性又能兼顾系统的实时性的调度策略呢？

通过对 FVM 指令执行过程的分析，不难看出，如果能预先知道调度时刻，并且能把调度时刻精确控制在 FVM 每次跳转执行下一条 Forth 指令时，即与 FVM“心跳”频率同步，便能破解上述难题，为嵌入式领域的许多特定应用提供更高性能的解决方案。正因为如此，FVMOS 多任务调度策略已成为 Forth 领域研究的热点，其中最具代表性的是 Forth 公司推出的 SwiftOS。SwiftOS 采用非抢占式策略，通过信号量和全局变量来支持进程间同步。在特有的交互调试方面，采用 Cross-Target Link 来支持各种终端调试接口。在手持设备的应用实验中，SwiftOS 能在多任务调度空循环的一个周期内，通过修改内核将设备置于休眠状态，比以往不可更改的二进制实时内核节省 70%的耗电量[4]。以商用的 SwiftOS 为代表，目前基于 FVM 的协同调度算法（cooperative scheduling algorithm，Round-Robin）已逐渐成为 Forth 系统的主流。

在物联网浪潮推动下，面对越来越复杂的嵌入式应用需求以及当今嵌入式操作系统研究领域亟待解决的重构、移植、可信、多核、众核等诸多难题[5]，FVMOS 再次进入了人们的视野。与直接管理 CPU 资源的传统操作系统相比，作为一种全新的操作系统，FVMOS 在体系结构、原理机制、调度策略上都存在明显的不同，其实时性尚处于“粗粒度”水平，还缺乏必要的理论研究和指导。本文在主流 FVMOS 多任务调度策略研究的基础上，在确保重构、扩展、移植、交互等固有特性的前提下，以提高实时性的算法优化为目标，充分挖掘架构在堆栈机上的独特性，提出一种高效精简的基于 Forth 虚拟机架构的嵌入式多任务操作系统调度策略。

1　FVMOS 系统结构

FVMOS 系统结构如图 1 所示。Flash 中存放在 FVM 上运行的 Forth 代码，FBS 底层是 Code 定义，上层是高级定义，在其之上是由高级定义组成的 FVMOS。由 FVMOS 可以创建隶属 stask 的若干用户任务。RAM 中有若干与用户任务相对应的由任务控制块 TCB、返回栈 RS、数据栈 DS 等组成的用户变量区，以及文本输入缓冲区 TIB、其他用户变量区和普通变量区等存储项。逻辑上，FBS、FVMOS 都可以进行 RAM 操作（细箭头），但物理上，系统的存取操作被封装在 FBS 的底层 Code 定义中，实际操作是透过这些分类存取指令完成的（粗箭头）。与抢占式 Stand Alone Forth88 不同，协同式 FVMOS 支持终端任务、后台任务和中断任务三种任务类型，虽然三种任务的 TCB 内容有差异，但都连接到多任务循环链表中（右虚线）。

FVMOS 架构应当在 Forth 核心词典和主控循环 Quit 之上，这与直接在寄存器处理器上架构操作系统有着明显的差异。若仅考虑单核 Forth 虚拟机，其可选的实时多任务操作系统实现方式为：利用 Forth 自身的特点，直接用汇编、C 语言或 Forth 自生成器（可异构）生成一个包含 Forth 虚拟机的基本系统 FBS（forth basic system），也就是 stask（system task，第一个终端任务），之后在其上定义封装：内存管理、任务管理、设备管理、错误陷井等 FVMOS 原语构件。若存储在 Flash 中的 FBS 和 FVMOS 是可重入的，那么新创建的任务 Utask（user task）只在 RAM 用户变量区中建立任务数据空间（运行在一个 Forth 虚拟机之上，适合于多核 SMP 模式），否则就需要复制 stask 到新的 Flash 和 RAM 工作区（运行在多个虚拟机之上，适合于多核 AMP 模式或多核离散模型）。同时，初始化 Utask 的任务控制块 TCB，并将新的 TCB 插入到系统的 TCB 链表中，进入多任务调度。

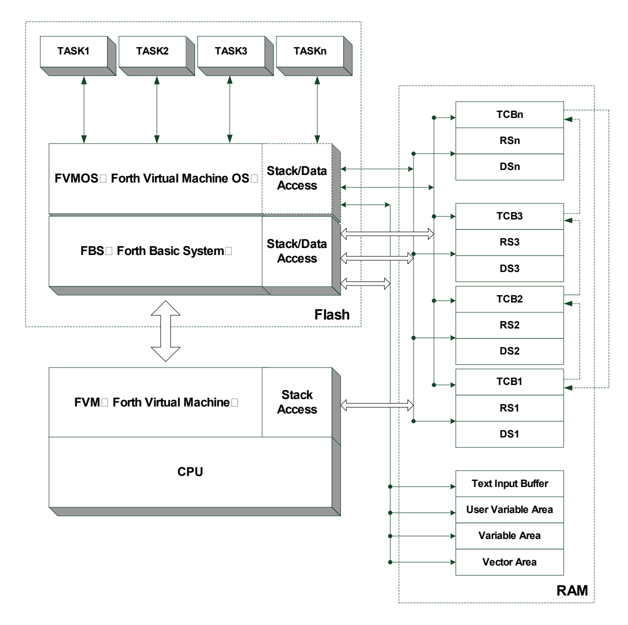


图 1　FVMOS 系统结构

2　基于 FVM 的任务控制块

与其他多任务调度方式不同，FVMOS 的协同式调度是基于虚拟机的，支持终端任务、后台任务和中断任务三种任务类型，在上述内存管理方式下，现场保护仅需要将当前返回栈指针 RP 压入数据栈，并将当前数据栈指针 SP 保存到该任务的用户变量区里。而恢复现场仅需要从该任务的用户变量区里恢复 SP，并将栈顶值存入 RP 指针。FVMOS 的任务控制块（task control block，TCB）就是用户变量区里与多任务调度有关的一块特殊区域，其结构如表 1 所示。

TCB 表中每一项都是通过用户变量 USER 定义的，其中基本表项（前六项）是每个任务的必备项，而附加表项是专门针对终端任务而设置的。附加表项（I/O 驱动）存放的是该终端任

务的 I/O 驱动向量，与该任务具体连接的终端设备有关，附加表项（解释器操作）存放的是该终端任务文本解释器与状态有关的操作向量。需要特别指出的是，TCB 没有保留返回栈指针，而是将当前返回栈指针放在数据栈栈顶。

表 1　基于 FVM 的 TCB

序号	偏移	名称	描述	类别
1	0	status	任务状态的 xt，UP 指针	
2	2	follower	下一个任务的 TCB 首址	
3	4	rp0	返回栈栈底指针	基本表项
4	6	sp0	数据栈栈底指针	
5	8	sp	数据栈栈顶指针（TOS）	
6	10	CATCHER	错误陷阱	
7	12	……	向量定义	附加表项（I/O 驱动）
……	……	……	向量定义	附加表项（解释器操作）

3　基于 FVM 的任务调度算法及实现

下面重点讨论基于 FVM 的多任务协同式调度算法，针对 Forth 语言和 Forth 系统的特点，算法描述采用了 Forth2012 标准。

3.1　任务调度

通常，轮询调度算法是在 TCB 的 STATUS 单元里保存任务的状态，每次任务调度时都去循环搜索处于 WAKE 状态的任务，但在基于 FVM 系统中，通过引入 Forth 向量字 PASS 和 WAKE，可以使调度算法进一步优化到省去状态比较和循环跳转。这时，TCB 里的 STATUS 存储的就不再是任务状态，而是可随时置换的 PASS 和 WAKE 向量字。轮询调度算法如下：

```
: MULTITASK-SCHEDULE
RP@  /当前返回栈指针 rp0（指向 PAUSE 后断点的 xts）压数据栈
SP@  /得到当前数据栈指针 sp0
SP  /取 tcb0[sp]地址
!  /将当前数据栈指针保存到当前任务的 tcb0[sp]
FOLLOWER  /得到 tcb0[follower]地址
@  /得到下一个任务的 tcb1
DUP @  /得到下一个任务的 status1（xts）
I-CELL+  /得到下一个任务 status1 里 pass 或 wake 过程的 pfa
>R  ;   /将下一任务 status1 里 pass 或 wake 的 pfa 压入返回栈
```

>R 之后，FVM 虚拟机开始跳转去执行 PASS 或 WAKE，当退出 MULTITASK-SCHEDULE 时，FVM 又开始返回到 PAUSE 后面的第一个字。

3.2　PASS 和 WAKE 过程

若是 PASS，rp^0 和 tcb^1 仍在数据栈里。PASS 过程将不断循环，跳过处于 PASS 状态的任务，直到找到 WAKE 为止。具体过程如图 2 所示。PASS 过程定义算法如下：

```
/ rp0 tcb1 -- n：多任务循环链表里处于 PASS 状态的任务数
:NONAME
CELL+  /得到存有第 n 个任务 followern 的地址
@     /得到第 n+1 个任务的 tcb n+1
DUP
@     /得到第 n+1 个任务的 status n+1（xts）
I-CELL+ /得到第 n+1 个任务 status n+1 里 PASS 或 WAKE 过程的 pfa
>R  ;  /将第 n+1 个任务 status n+1 里 PASS 或 WAKE 的 pfa 压入返回栈
CONSTANT PASS  /把无名定义的 xts 存在 PASS 常数里， PASS 执行上述无名定义
```

若是 WAKE，rp^0 和 tcb^m 仍在数据栈里。WAKE 过程将恢复处于 WAKE 状态任务的数据栈和返回栈，并由 FVM 实现跳转。WAKE 过程定义算法如下：

```
/  rp0 tcbm -- m：扫描多任务循环链表时处于 wake 状态的第一个任务
:NONAME
UP!  / 将用户变量区指针指向第 m 个任务（WAKE 状态）的 tcbm
SP   / 获取第 m 个任务的 tcbm[sp]地址，此时数据栈已切换到第 m 个
/ 任务的数据栈，每个任务的数据栈的栈顶都存有该任务的 rpm
@   / 得到第 m 个任务数据栈指针 spm
SP!  / 激活第 m 个任务的数据栈
RP!  ;  / 激活第 m 个任务的返回栈
CONSTANT WAKE  / 把无名定义的 xts 存在 WAKE 常数里，
               / WAKE 执行上述无名定义
```

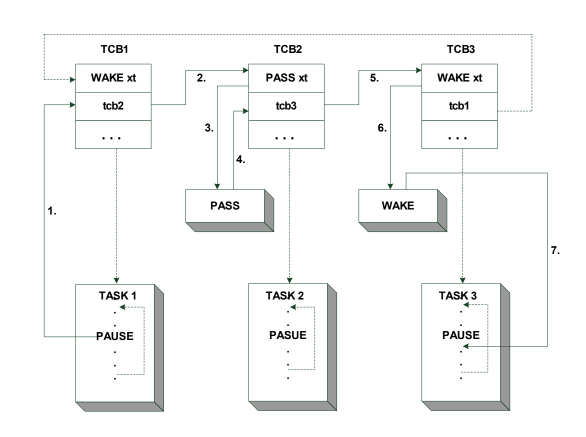


图 2　PASS 和 WAKE 过程

3.3　任务控制

多任务调度器 PAUSE 可以由 SINGLE 和 MULTI 定义来开关，SINGLE 将多任务调度器 PAUSE 设为空操作，关闭多任务调度，仅保留一个终端任务；MULTI 将多任务调度器 PAUSE 指向 MULTITASK-SCHEDULE，启动多任务调度。SINGLE、MULTI 定义、任务状态控制算法如下：

```
: SINGLE
```

```
['] NOOP IS PAUSE ;  / 将 PAUSE 设为空操作，仅保留一个终端任务
: MULTI
 ['] MULTITASK-SCHEDULE IS PAUSE ; / 将 PAUSE 指向 MULTITASK-SCHEDULE
: TASK-STOP
PASS STATUS ! PAUSE ;  / 进入多任务调度，即刻停止当前任务
: TASK-SLEEP
PASS SWAP ! ;          / 在下一次多任务循环中让 tcb 任务休眠
: TASK-WAK
WAKE SWAP ! ;          / 在下一次多任务循环中唤醒 tcb 任务
```

在多任务系统启动之前，在任务创建和初始化之后，需要对每个任务进行定义。ACTIVATE 把紧随其后的 Forth 命令设置为当前 tcb 所指向任务的任务体，其定义算法如下：

```
/ tcb --
: ACTIVATE
DUP
6 + @ CELL-  / 得到 tcb 任务准备压入第一个数据的数据栈指针
OVER
4 + @ CELL-  / 得到 tcb 任务准备压入第一个数据的返回栈指针
R> OVER I-CELL+  !  / 将 ACTIVATE 后的断点压入 tcb 对应任务的返回栈
OVER !  / 将 rp 压入数据栈，为 WAKE 过程做准备
OVER 8 + !  / 将数据栈指针保存到 tcb [sp]
TASK-WAKE ;
```

以定义在主任务中的 SCAN 为例，如图 3 所示，当主任务执行时，ACTIVATE 将后面的 BEGIN……AGAIN 循环定义为 tcb 所指任务的任务体，而不是去执行这个循环。只有当 tcb 指向的任务（从任务）执行时，才去执行这个循环（任务体）。由此可见，SCAN 定义被隐形的分解为主任务和从任务执行的两个部分。

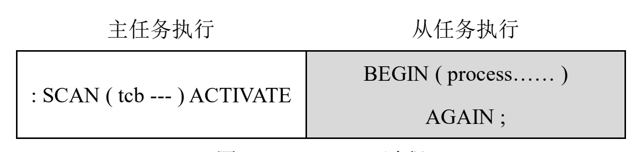


图 3　ACTIVATE 过程

3.4　后台任务

后台任务适用于不需要连续串行 I/O、不需要连接终端，甚至不需要解释/编译器的场合，其任务创建、初始化、控制等与 TASK、TASK-INIT、ACTIVATE 、ADD-NEXTTASK 等过程相同。

3.5　终端任务

Forth 系统是一种独特的“编译/解释”双态系统，支持终端交互式操作，为在线开发、调试、更新、修改带来了便利。正因为如此，早期小型机 Forth 系统（如 PDP-11 Stand Alone Forth-11）中支持多个终端用户和多个终端任务的特性也被继承下来，即使在资源有限的嵌入式系统环境下，Forth 系统依然保留了这一优势特点，并赋予每个终端任务不同的工作内容。

与后台任务相比，终端任务为支持串口和解释器工作，可以在每个任务的用户变量区里附加更多的操作信息，如果目标系统里有一个解释器，那么其对应的终端任务就需要依据不同的串口硬件，在用户变量区里建立终端任务的 I/O 驱动和与文本解释器输入缓冲区有关的操作。

终端任务的创建与后台任务基本相同，都是调用 TASK 过程，所不同的是申请追加分配的用户变量区可以包含附加的 I/O 驱动向量、解释器输入流操作向量，以及输入流缓冲区。终端任务的初始化除完成 TASK-INIT 过程外，还需要针对不同的终端类型，填写或置换字符、回车、光标等公共操作的执行向量。需要说明的是，系统上电时，核心系统缺省就已经包含了一个终端任务，此时多任务循环链表里只用一个任务指向该任务自身的 TCB。

3.6　中断任务

与抢占式的强实时 OS 不同，为减少开销，同时兼顾到性能，基于 FVM 的多任务调度并不直接使用中断进行任务切换，而是依然将任务切换放在 Forth 定义之间。中断发生时，如果有对应的中断任务，待处理完中断事务后，中断服务程序将执行与 TASK-WAKE 类似的操作，使该任务在下一次调度（PAUSE）中启动运行，如图 4 所示。由此可见，这种中断事件驱动方式是一种弱实时，要经历若干可预见的 Forth 定义才能获得响应，但由此换来的是系统简单、良好性能的保证，不论中断何时发生，启动中断任务的时刻是可以预知的，只能发生在有 PAUSE 语句的地方。在这种方式下，中断任务的创建和初始化与后台任务相同，其使用具有较大的灵活性。

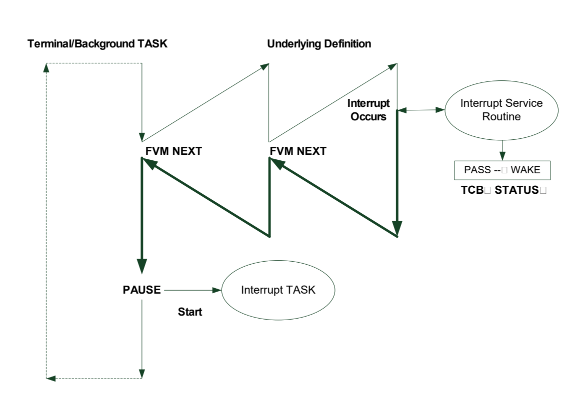


图 4　中断调度过程

一般来说，除了特殊的周期性定时任务，中断任务往往都需要在每次中断发生时从头至尾一次完成（没有 PAUSE 过程），而非断点处运行。此外，若要提升系统的实时性，允许存在多个优先级的中断任务，并且具有抢占性，那么就需要对上述的多任务组织、调度进行改进。解决方案之一就是在协同多任务循环链表之外，再构建一个抢占多任务链表（中断任务链表），在 STATUS、FOLLOWER 基础上增加中断号（INT-NO）和优先级（PRIORITY）等 TCB 表项。同时，使用与 PAUSE、PASS、

WAKE 等类似的过程，去搜索处于唤醒状态优先级更高的任务，并通过 WAKE 过程启动中断任务。

4　实验评估

在 Arduino 嵌入式硬件平台上，借助开源的 Forth 虚拟机，实现了上述算法，并设计了一个含有终端任务、后台任务和中断任务等三种任务类型的多任务应用程序，实验代码如下所示：

```
: MS ( n -- ) PAUSE 0 ?DO 1MS LOOP ;  / 每隔 n 毫秒调用 PAUSE 进行等待
VARIBLE M  / TASK1 中 TASK2 使用的全局变量 M
VARIBLE N  / TASK1 中 TASK3 使用的全局变量 N
: INIT ( -- ) 0 M ! 0 N ! ;
$40 $40 0 BACKGROUND-TASK TASK2  / 建立后台任务 TASK2，在 FLASH
/ 中创建任务头，分配用户变量区
: TASK2-BODY ( -- ) BEGIN 1 M +! &10 MS AGAIN ; / 定义 TASK2 任务体
$40 $40 0 INTERRUPT-TASK TASK3  / 建立中断任务 TASK3，在 FLASH 中
/ 创建任务头，分配用户变量区
: TASK3-BODY ( -- ) BEGIN 1 N +! TASK-STOP AGAIN ;/定义 TASK3 任务体
: STARTTASKER ( -- )  / 初始化并启动多任务
TASK2 BACKGROUND-INIT  / TASK2 初始化，在 RAM 中创建 TCB2
TASK3 INTERRUPT-INIT  / TASK3 初始化，在 RAM 中创建 TCB3
TASK2 TIB>TCB ACTIVATE TASK2-BODY  / 连接 TASK2 任务体
TASK3 TIB>TCB ACTIVATE TASK2-BODY  / 连接 TASK3 任务体
ADD-FIRSTTASK  / 建立 TCB1 循环链表
TASK2 TIB>TCB ADD-NEXTTASK  / 将 TCB2 加入循环链表
TASK3 TIB>TCB ADD-NEXTTASK  / 将 TCB3 加入循环链表
MULTI ;  / 启动多任务
: TASK-TURNKEY ( -- )  / 上电启动向量
APPLTURNKEY INIT STARTTASKER ;
```

在第一个终端任务 TASK1（STASK）里定义后台任务 TASK2 和中断任务 TASK3。多任务启动后，TASK1 在 TASK2 任务体每次调用 MS 时执行一次，平均间隔 10 ms；TASK2 在 TASK1 每次等待键盘输入时执行；TASK3 一直处于休眠状态（PASS），直到有对应的中断发生，状态变为唤醒（WAKE），在距离最近的 PAUSE 处开始执行（这段距离就是中断延迟），执行后再次进入休眠状态。实验表明，系统运行稳定可靠，在具备高效的在线交互和后台处理能力的同时，具备一定的实时处理能力。协同式的特点简化了任务间繁琐的资源互斥访问机制，通过 TASK1 终端任务，可以随时跟踪到 M、N 及各任务 TCB 的变化。

在时间片轮转调度系统中，系统响应时间一般可表示为

$$SRT=\sum_{i=0}^{N}(TSi+TST) \tag{1}$$

其中：*TSi* 为各任务时间片；*TST* 为任务切换时间；N 为并发任务数。若采用带优先级的算法，并且各任务的时间片与优先级成正比，则

$$SRT=TSmin\times M\sum_{i=0}^{N}(Pi+1)+N\times TST \tag{2}$$

其中：M 为比例因子；TSmin 为最小时间片；*Pi* 为各任务优先级。当 *Pi* 取值 $0\sim N-1$ 且 M＝1 时，则

$$SRT=TSmin\frac{N(N+1)}{2}+N\times TST \tag{3}$$

为保证最小时间片时，系统最大开销不超过 5%，设 $TST/TSmin=0.05$，则

$$SRT=N\times(10N+11)\times TST \tag{4}$$

若不考虑优先级调度，即 $Pi=0$、$TSi=TS$，上述算法就退化为相等时间片轮转调度，则

$$SRT=N(TS+TST) \tag{5}$$

同样，为保证系统平均开销不超过 1%，设 $TST/TS=0.01$，则

$$SRT=101\times N\times TST \tag{6}$$

本文所述算法的系统响应时间可用式（6）来近似表示，所不同的是基于 FVM 的任务切换要比基于 CPU 的快很多，仅仅只需保留或恢复 SP 指针就能实现。与文献[2]的 Stand Alone Forth88 并发任务调度相比，基于 FVM 的任务切换代码（机器码）规模只有其 1/10，因此，同等开销下，系统响应时间缩短为

$$SRT=10.1\times N\times TST \tag{7}$$

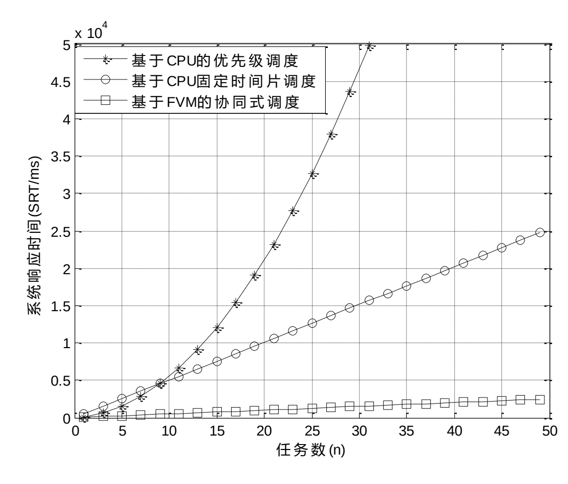


图 5　三种调度方式的系统响应时间

基于 CPU 的优先级和固定时间片调度以及基于 FVM 协同式调度的系统响应时间如图 5 所示。可见，本文提出的基于 FVM 的调度算法，当任务数递增时（可为后台或中断任务类型），任务切换和在线交互无明显的延迟。此外，每个任务的时间片大小都可由用户依据应用需求灵活决定，既可以调整任务体大小，又可以在各自任务体内通过增加或减少 PAUSE 改变调度

的精细“粒度”，而不必遵循系统的预定设置，即 *SRT* 完全可以由用户自行设计。

5 结束语

现在主流的 FVMOS 的多任务调度依然采用的是协同式调度器（cooperative scheme，Round-Robin），也就是说每个任务的启动时间都是预先确定好的。尽管抢占式多任务系统实时性很强，但若引入 FVMOS，势必会影响到在线交互终端任务的及时响应和有序执行，而在线交互又是 Forth 界长期引以为豪而不肯放弃的特性。实际上，两者的取舍与嵌入式应用环境有关。研究表明，嵌入式系统往往有非常重的 CPU 负载，而抢占式调度器的开销将明显高于基于虚拟机的协同式调度器，在一个负载非常重的系统中，协作调度器的简单性和性能比抢占式调度器更高。FVMOS 虽然只是一个轻量级的协同式调度器，但依然可以借助 Forth 灵活简单的特点，对实时事件作出快速处理。据报道，一台四轴炸弹处理机使用标准的协同式调度器运行了 12 个任务，在其他复杂应用领域甚至还有多达 400 个任务的应用报道。

与传统 OS 的多任务调度算法不同，基于 FVM 的调度代码都是 Forth 高级定义，没有涉及 CPU 的任何细节，通过 FVM 的抽象做到了与硬件无关，平台之间的移植可以不必更改一行代码，从而使系统天然具备了可重构、可扩展、可移植的特性。秉承 Forth 思想，“不满意就去修改它”，为进一步提高系统的实时性，这种框架和算法的改进可以不断进行下去。例如，中断任务的上下文切换时间可以从任务体层级下降到中断服务程序退出之时，但这样做所带来的复杂性和开销还需要进一步的研究和评估。

参考文献：

[1] McGuire T E. Kitt peak multi-tasking FORTH-11 [J]. The Journal of Forth Application and Reseach, 1984, 2 (2): 57-67.

[2] 代红兵. 高效微机实时多任务操作系统设计与实现 [J]. 中国科学院研究生院学报, 1993, 10 (3): 283-292.

[3] Moore C. colorForth [EB/OL]. (2009) . https://colorforth. github. io/cf. htm.

[4] Forth Inc. SwiftX cross compilers for embedded systems applications [EB/OL]. (2016) . https://www. forth. com/embedded/.

[5] 杨霞. 高可信嵌入式操作系统体系架构研究 [D]. 成都: 电子科技大学, 2010.

[6] Pelc S. Programming Forth [M]. Southampton: MicroProcessor Engineering Limited, 2011: 97.

[7] 代红兵, 杨为民, 王丽清, 等. 多目标Forth自生成器的研究与实现 [J]. 计算机应用研究, 2014, 31 (4): 1109-1114.

[8] 杨为民, 代红兵, 安红萍, 等. 一种新的嵌入式 Forth 实时操作系统的研究 [J]. 云南大学学报: 自然科学版, 2013 (S2): 96-103.

[9] Frenger P. Forth and AI revisited: BRAIN. FORTH [J]. ACM SIGPLAN Notices, 2004, 39 (12): 11-16.

[10] 代红兵. 新型、高效微机 Forth 语言的研制 [J]. 中国科学院研究生院学报, 1993, 10 (1): 62-69.

[11] FORTH Inc. Featured forth applications [EB/OL]. (2009) . http://www. forth. com/resources/app Notes.

[12] IntellaSys, A TPL Group Enterprise. SEAforth 40C18 scalable embedded array processor [EB/OL]. (2008) . http://www. intellasys. net/templates/trial/ content/SEK _40C18_DataSheet_1. 1. pdf.

[13] Ieee B E. IEEE standard for boot (initialization configuration) firmware: bus supplement for IEEE 896 (futurebus+) [S]. 2002: i.

[14] The Forth Interest Group. Forth compilers page [EB/OL]. (2009) . http://www. forth. org/compilers. html.

[15] Miller F P. Colorforth [M]. 2010: 28.

[16] Hanna D M, Jones B, Lorenz L, et al. An embedded Forth core with floating point and branch prediction [C]// Proc of IEEE, International Midwest Symposium on Circuits and Systems. 2013: 1055-1058.